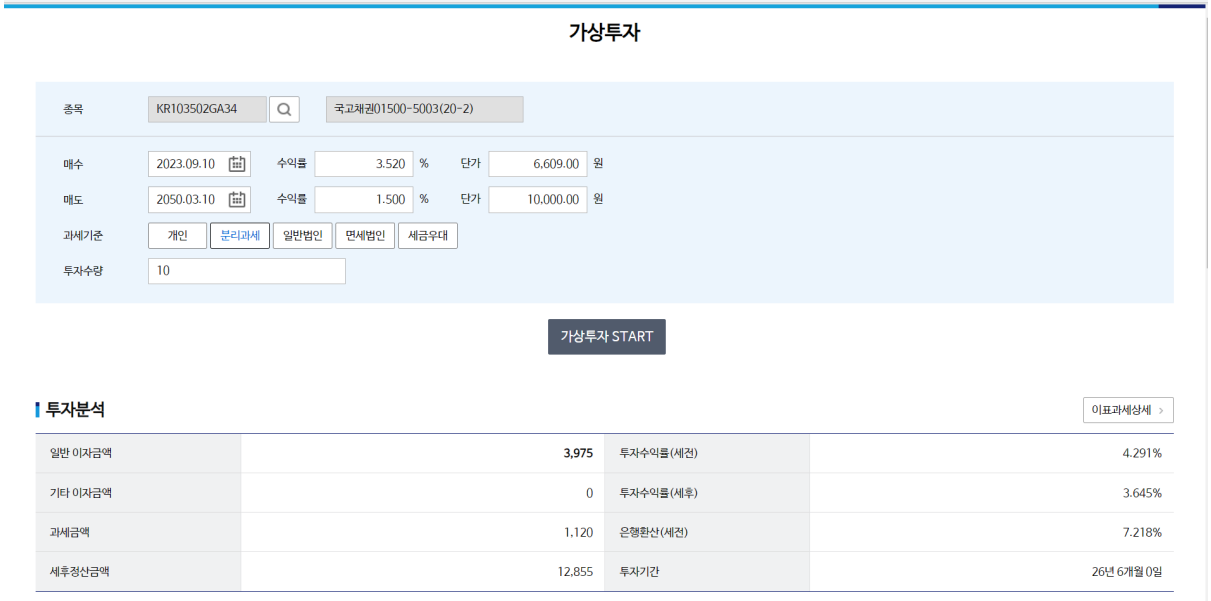
신한투자증권 채권 가상투자 서비스에 디자인 패턴 적용하기

```
21시 고객님이 투자하신 bonds.SimpleBond@4edde6e5 리포트입니다
단리 이자: 10.0
세전 원금액: 0.0
수익률: -45.0%
만기까지 남은 기간 (일): 9607
투자 기간 (일): 9607
연간 이자 지급액: 100.0
장내 채권 시장은 휴장 시간입니다. 이 시간에는 채권 거래를 할 수 없습니다.
장 운영에 관해 문의 사항이 있으신 경우 신한알파의 챗봇 서비스를 이용해주세요.
```

> 이번 글에서는 신한투자증권 채권 가상투자 서비스에 디자인패턴을 도입하여 재구성하게 된 배경과 그 과정을 담아내고자 합니다.

배경



기존에 증권투자권유대행인을 준비하면서 주식과는 다른 매력을 갖고 있는 채권 시장에 대한 관심이 생겼습니다. 그러나 저는 채권 시장을 실제로 이용한 경험이 없어서 채권에 대한 이해도가 주식보다 다소 낮았습니다.

이와 같은 저의 배경을 통해, 신한투자증권을 이용하는 고객들 중에서도 채권에 대한 이해도가 낮아 채권에 쉽게 진입하지 못하고 있는 잠재고객들이 많다는 생각을 했습니다. 따라서 저는 채권의 기본적인 정보와 시스템을 한번에 알려주는 **가이드 서비스에 대한 필요성**을 느끼게 되었습니다.

또한 채권 시장을 공부하면서 느낀 점은 채권의 종류, 계산 방법, 채권시장 운영 시간 등이 **디자인 패턴과 유사한 형태**를 가지고 있다는 것이었습니다. 이러한 유사성을 이용해 채권 시장을 이해하고 개발하는데 도움이 될 수 있는 **디자인 패턴을 도입**하고자 했습니다.

그래서 저는 기존에 있는 **신한투자증권의 채권 가상투자 서비스**를 디자인패턴을 통해 기능을 추가하고 개선하기 위한 프로젝트를 고안하게 되었습니다.

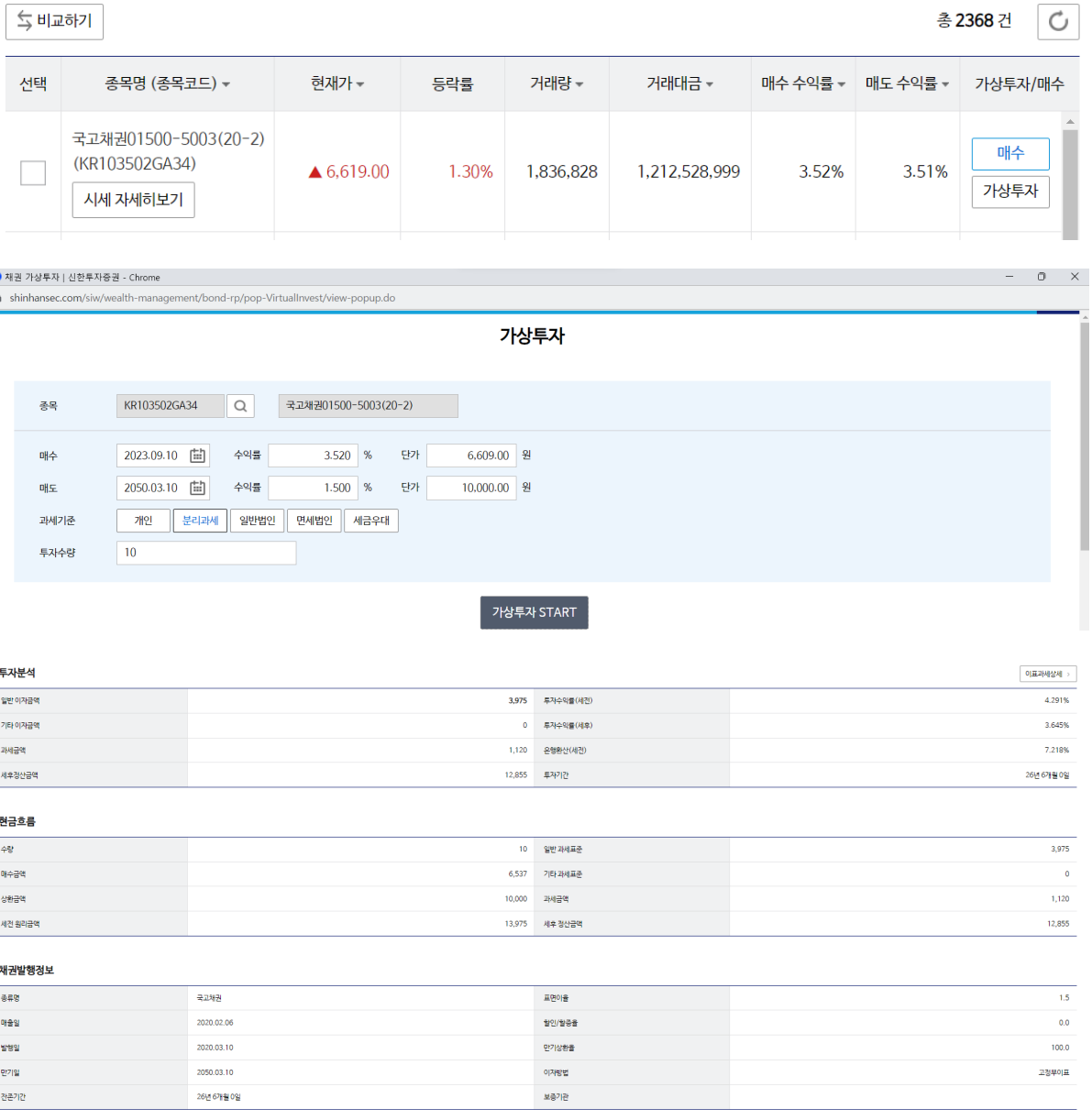
이 프로젝트를 통해 기존 서비스에 디자인 패턴을 적용하여 향후 채권에 대한 더 많은 가이드를 제공하고 싶을 때(혹은 채권 관련 법률이 수정되어 재정비가 필요할 때) 재사용이 용이할 수 있도록,

사용자들이 위 서비스를 통하여 채권 시장을 더 쉽게 이해하고 활용할 수 있도록 돕고자 했습니다. 이를 통해 고객들에게 더 나은 투자 경험을 제공하고, 채권 시장에 대한 이해도를 높일 수 있는 기회를 제공하려고 합니다.

이번 리팩토링 프로젝트를 통해 채권 시장에 대한 이해도를 높이고, 신한투자증권의 채권 가상투자 서비스를 더욱 현대적이고 사용자 친화적인 방향으로 발전시켜 채권 투자에 대한 잠재고객을 끌어올 수 있기를 희망합니다.

기존 채권 가상투자 서비스 구성 및 흐름 분석

신한투자증권 채권 가상투자 서비스 구성 및 흐름은 다음과 같습니다.



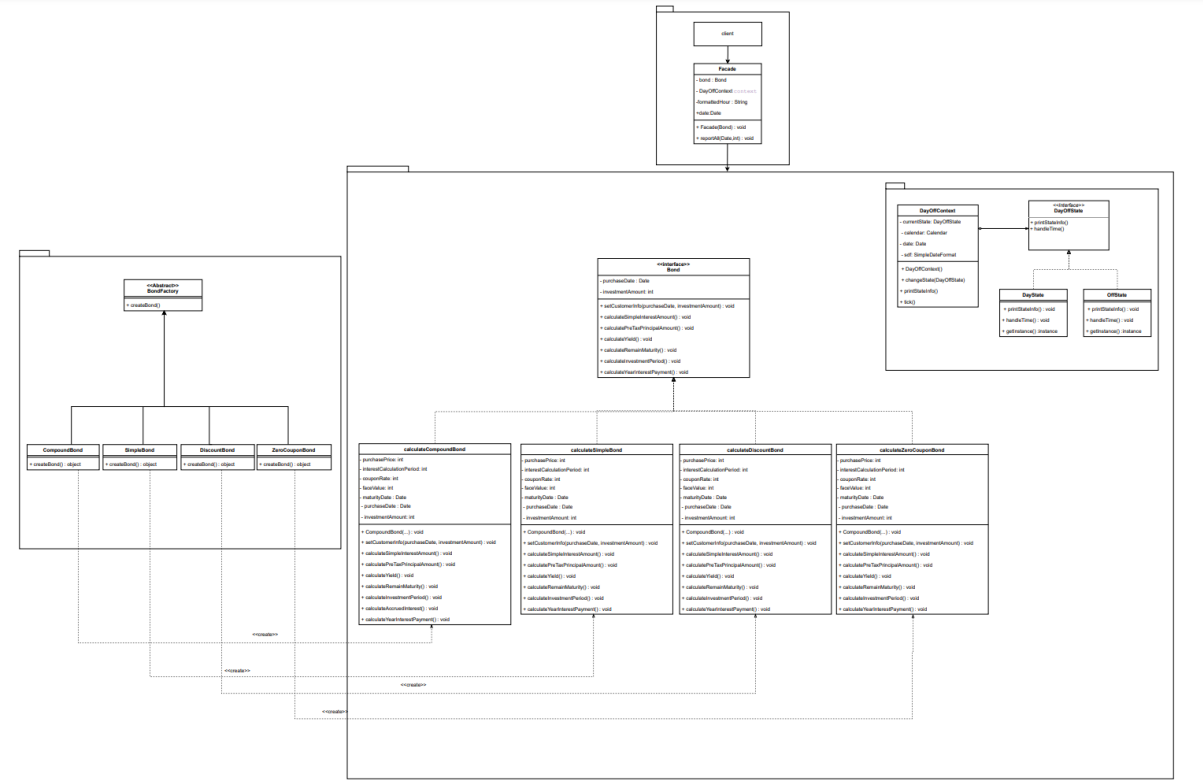
1. 우측 가상투자 버튼을 누르면 자동으로 가상투자 팝업이 생성되며, 종목 코드와 종목명이 입력이 된다.

2. 이에 해당되는 매수, 매도일자, 수익률, 단가가 적용된다.
3. 사용자가 과세기준을 선택하여 넣고 투자 수량을 입력한 후, 가상투자 START 버튼을 누르면 투자분석, 현금흐름, 채권발행 정보를 계산해서 알려준다.

수정 및 고려 사항

다음은 위와 같은 신한투자증권의 채권 가상투자 서비스 중 중점을 두어 저만의 방식으로 재구성해본 고려 사항입니다.

1. **State 패턴**을 사용하여 장내채권 매매시간(평일 오전 9:00 ~ 15:30) 상태에 따른 다른 대응 적용
2. 채권의 이자 및 원금지급방법(복리채, 단리채, 할인채, 이표채)에 따라 채권 정형화 한 뒤, 채권 생성시 **Factory Method 사용**
3. **Facade**를 통한 고객 채권 투자 리포트 생성 간편화



화질이 안좋은 관계로 아래 PDF파일 열람을 권장합니다.

최종.drawio (7).pdf

도입과정

1. State 패턴을 사용하여 장내채권 매매시간 (평일 오전 9:00 ~ 15:30) 상태에 따른 다른 대응 적용

State Pattern

💡 상태 디자인 패턴은 프로그램의 객체 내부의 상태에 따라 동작을 변경해야할 때 사용하는 디자인 패턴입니다.

상태 패턴을 사용하는 간단한 실생활 예시

예를 들어, 자판기는 금액이 투입되고 금액이 부족하냐, 충분하냐에 따른 상태에 따라 음료수를 판매하는데,

이때 상태패턴을 사용하면 자판기의 상태에 따라 음료수를 판매하는 동작을 변경할 수 있습니다. 만약 상태가 동전이 충분한 상태가 된다면 자판기는 음료를 뽑을 수 있는 권한을 줍니다!

또한 시간에 따른 상태를 생각해보면, 여러분들이 종사하고 있는 증권회사의 근무시간을 들어볼 수도 있습니다.

8시부터 5시까지 우리는 “근무 상태”에 있어서 근무에 관련된 작업을 하지만, 이 외의 시간에는 “휴식 상태”에 들어가서 최대한 휴식을 취하려고 하고있죠. 이게 바로 State 패턴입니다.

State 패턴의 장점

State 패턴의 장점은 다음과 같습니다.

- 하나의 객체에 대한 여러 동작을 구현해야 할 때 상태 객체만 수정하므로 **동작의 추가, 삭제 및 수정이 간단**해집니다.
- State 패턴을 사용하면 객체의 상태에 따른 조건문(if/else, switch)이 줄어들어 **코드가 간결해지고 가독성**이 올라갑니다.

따라서 이러한 객체의 상태에 따른 구현이 간결해지고 가독성이 올라간다는 장점 때문에,

저는 상태패턴을 채권 시장의 운영시간에 따른 상태에 적용해보았습니다.

State 패턴을 사용한 채권 매매 가능여부 출력 설계

채권 시장은 **평일 오전 9:00부터 오후 3:30까지** 운영됩니다. 이 시간 이외에는 채권 매매가 불가능합니다. 이와 같은 특성을 고객들에게 알리고자, State 패턴을 이용하여 현재 채권시장운영 여부를 알려주는 구조를 설계해보았습니다.

Before

```
public class DayOffContext {
    private boolean isMarketOpen;

    public DayOffContext() {
        isMarketOpen = isMarketOpen();
    }

    public void tick() {
        isMarketOpen = isMarketOpen();
    }

    public void printStateInfo() {
        if (isMarketOpen) {
            System.out.println("현재 장내 채권시장 운영 시간입니다. 자세한 문의사항은 지점을 방문해주세요");
        } else {
            System.out.println("장내 채권 시장은 휴장 시간입니다. 이 시간에는 채권 거래를 할 수 없습니다.");
        }
    }

    private boolean isMarketOpen() {
        // 현재 시간(시)을 얻는 코드로 대체
        int currentHour = getCurrentHour();

        // 시간대에 따라 채권 시장의 운영 여부 판단
        if (currentHour >= 9 && currentHour < 17) {
            return true; // 채권 시장 운영 중
        } else {
            return false; // 채권 시장 휴장 중
        }
    }

    private int getCurrentHour() {

        return 0;
    }
}
```

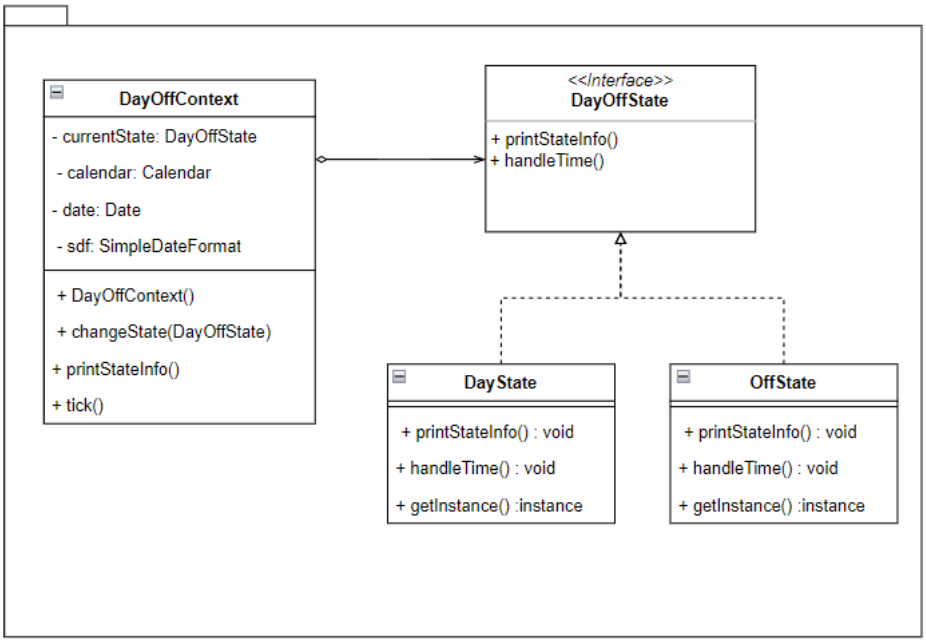
장내 채권시장 운영에 따른 상태 변화를 조건문으로 만들 수 있고, 사실 이러한 방법이 당장 만들기엔 편하고 쉽지만

이러한 접근 방식은 확장성과 유지 보수 측면에서 좋지 않을 수 있습니다.

따라서 코드의 가독성과 유지보수성을 위해 State Pattern을 사용했는데, 특히나 객체 지향 프로그래밍에서는 상태 패턴(State Pattern)을 사용하여 이러한 상황을 더 효율적으로 다룰 수 있다. State Pattern은 객체의 상태에 따라 객체의 행동을 변경하고 확장하기 위한 디자인 패턴 중 하나입니다.

After

State 패턴을 적용한 UML클래스 다이어그램은 다음과 같습니다.



1. **DayOffContext 클래스:**
   - `DayOffContext` 클래스는 채권 시장의 운영 상태를 추상화하고, 현재 시간에 따라 다른 상태를 유지하는 클래스입니다.
   - 이 클래스는 클라이언트에게 채권 시장의 운영 상태를 알려주고, 현재 시간에 따라 상태를 변경하므로 채권 거래 시간과 휴장 시간을 관리하기 용이합니다.
2. **DayOffState 인터페이스:**
   - `DayOffState` 인터페이스는 `DayOffContext` 클래스의 상태를 정의하는 인터페이스입니다. 이를 구현하는 클래스는 채권 시장의 상태를 나타내는데 사용됩니다.
   - 이 인터페이스를 통해 `DayState`, `OffState` 클래스와 같이 시간에 따른 서로 다른 시장 상태를 구현할 수 있으므로 채권 시장의 운영 상태를 확장하거나 변경하기 쉽습니다.
3. **DayState 클래스:**
   - `DayState` 클래스는 채권 시장이 개장한 상태를 나타내는 클래스입니다.
   - 이 클래스는 시장이 개장한 상태에서 클라이언트에게 정보를 제공하고, 시장 운영 시간 내에서만 거래를 허용합니다.
4. **OffState 클래스:**
   - `OffState` 클래스는 채권 시장이 휴장한 상태를 나타내는 클래스입니다.
   - 이 클래스는 시장이 휴장한 상태에서 클라이언트에게 정보를 제공하고, 시장 운영 시간 외에는 거래를 제한합니다.

이러한 클래스와 구조를 사용하면 채권 시장의 운영 시간을 쉽게 관리하고, 클라이언트에게 시장 상태를 알려주며, 특정 시간대에 거래를 허용하거나 제한할 수 있습니다. 새로운 시장 상태가 필요한 경우 `DayOffState` 인터페이스를 추가적으로 구현하여 시장 상태를 확장하거나 변경할 수 있으므로 시장 운영 규칙을 쉽게 수정할 수 있습니다.
이와 같은 State Pattern을 통해 코드의 확장성이 향상되며, 복잡한 시스템에서 채권 거래 시간 관리가 용이해집니다.

---

2. Factory Method를 활용한 채권의 이자 및 원금지급방법(복리채, 단리채, 할인채, 이표채)에 따라 채권 정형화 설계

Factory Method 패턴 소개

Factory Method 패턴은 객체 생성을 캡슐화하는 디자인 패턴 중 하나입니다. 객체 생성을 처리하는 코드를 별도의 클래스나 메서드로 분리하여 객체 생성의 변화에 대비할 수 있습니다. 특정 기능의 구현은 개별 클래스를 통해 제공되는 것이 바람직한 설계입니다. 객체 생성 방식의 변화는 해당되는 모든 코드 부분을 변경해야 하는 문제가 발생하는데, Factory Method 패턴을 사용하면 이러한 문제를 해결할 수 있습니다.

Factory Method 실생활 예시

Factory Method 패턴은 실생활에서도 많이 사용됩니다. 예를 들어, 우리가 평소에 자주 마시는 커피에도 FactoryMethod가 들어갑니다.

커피 전문점에서는 커피를 만드는 방법에 따라 아메리카노, 라떼, 카푸치노 등 다양한 종류의 커피를 제공합니다.

이때, 아메리카노, 카페라떼 등 커피 종류에 따라 다른 클래스에서 객체를 생성하고, 공통된 인터페이스를 사용하여 커피를 제공하는 방식으로 Factory Method 패턴이 적용됩니다!

Factory Method 장점

Factory Method 패턴은 다음과 같은 장점을 가지고 있습니다.

- 객체 생성 코드를 별도의 클래스/메서드로 분리함으로써 객체 생성의 변화에 대비할 수 있습니다.
- 특정 기능의 구현은 개별 클래스를 통해 제공되는 것이 바람직한 설계입니다.
- 객체 생성 방식의 변화는 해당되는 모든 코드 부분을 변경해야 하는 문제를 해결할 수 있습니다.

이와 같은 Factory Method는 별도의 클래스/메서드를 통해 변화에 대비할 수 있기 때문에,
다중의 형태를 갖고있는 채권의 유형에도 적용할 수 있다는 장점이 있습니다.

Factory Method 적용하여 채권의 이자 및 원금지급방법에 따라 채권 정형화 후, 채권 생성

다양한 형태를 갖고있는 채권의 유형들을 Factory Method 적용하여 채권의 이자 및 원금지급방법에 따라 채권 정형화 후, 채권 생성하는 방식을 통해 구조화 시켰습니다.

Before

```
package bonds;

import java.util.Date;

public class BondFactoryWithoutMethod {
```

```
    public Bond createBond(String bondType, int purchasePrice, int interestCalculationPeriod, double couponRate, int faceValue, Date m
        Bond bond = null;

        switch (bondType) {
            case "CompoundBond":
                bond = new CompoundBond(purchasePrice, interestCalculationPeriod, couponRate, faceValue, maturityDate, purchaseDate, i
                break;
            case "SimpleBond":
                bond = new SimpleBond(purchasePrice, interestCalculationPeriod, couponRate, faceValue, maturityDate, purchaseDate, inv
                break;
            case "DiscountBond":
                bond = new DiscountBond(purchasePrice, interestCalculationPeriod, couponRate, faceValue, maturityDate, purchaseDate, i
                break;
            case "ZeroCouponBond":
                bond = new ZeroCouponBond(purchasePrice, interestCalculationPeriod, couponRate, faceValue, maturityDate, purchaseDate,
                break;
            default:
                throw new IllegalArgumentException("지원하지 않는 채권 종류입니다: " + bondType);
        }

        return bond;
    }

    public static void main(String[] args) {
        BondFactoryWithoutMethod bondFactory = new BondFactoryWithoutMethod();

        // 채권 생성 및 초기화
        String bondType = "CompoundBond"; // 채권 종류 선택
        Date purchaseDate = new Date(); // 예시: 구매일자 (현재 날짜)
        int investmentAmount = 1000; // 예시: 투자 금액

        // 채권 생성
        Bond bond = bondFactory.createBond(bondType, 10, 100, 10, 10, purchaseDate, purchaseDate, investmentAmount);

        // 채권 계산 메서드 호출
        bond.calculateSimpleInterestAmount();
        bond.calculatePreTaxPrincipalAmount();
        bond.calculateYield();
        bond.calculateRemainMaturity();
        bond.calculateInvestmentPeriod();
        bond.calculateYearInterestPayment();
    }
}
```

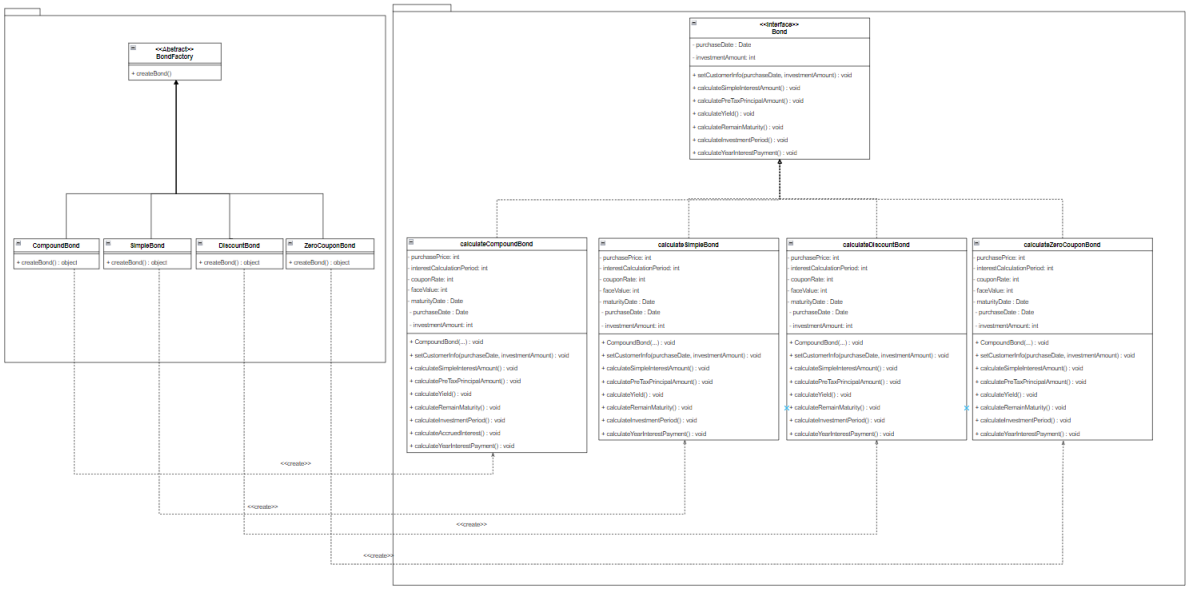
위에 보시는 코드처럼 Factory Method를 쓰지 않을 경우,
채권의 이자 및 원금지급방법에 따라 채권을 생성하는 코드에서는 조건문과 switch문이 자주 사용됩니다.

이로 인해 코드의 가독성이 떨어지고, 유지보수가 어려워진다는 특징이 있습니다. 또한, 위와 같은 코드는 채권이 각자 독립적으로 작동하지 않기 때문에 새로운 종류의 채권을 추가할 때마다 코드를 수정해야 하는 문제가 발생합니다.
 때문에 저는 Factory Method 디자인 패턴을 채권의 이자 및 원금지급방법에 적용하여 채권을 정형화하는 방식을 취했습니다.

After

Factory Method 패턴을 적용한 UML클래스 다이어그램은 다음과 같습니다.

(화질 관계상 첨부된 PDF를 보시길 권장드립니다.)



- **AbstractBondFactory (<<Abstract>>BondFactory):**
  - `AbstractBondFactory`는 채권 객체를 생성하는 추상 팩토리 클래스입니다. 이 클래스를 상속하여 구체적인 채권 종류에 대한 팩토리를 생성합니다.
  - 새로운 채권 종류를 추가할 때 Factory Method 패턴을 이용하여 해당 채권의 객체 생성 방법을 표준화할 수 있으며, 코드의 유지보수와 확장성을 개선합니다.

아래 4개는 AbstractBondFactory 를 상속받는 채권 클래스입니다.

1. **CompoundBondFactory 클래스:**
   - `CompoundBondFactory` 클래스는 복리 채권 객체를 생성하는 팩토리 클래스입니다. `BondFactory`를 상속하고 `createBond` 메서드를 구현합니다.
   - 새로운 복리 채권을 생성하려면 `CompoundBondFactory`를 사용하며, 복리 채권 객체 생성 로직을 캡슐화하여 단순화합니다.
2. **SimpleBondFactory 클래스:**
   - `SimpleBondFactory` 클래스는 단리 채권 객체를 생성하는 팩토리 클래스입니다. `BondFactory`를 상속하고 `createBond` 메서드를 구현합니다.
   - 새로운 단리 채권을 생성하려면 `SimpleBondFactory`를 사용하며, 단리 채권 객체 생성 로직을 캡슐화하여 단순화합니다.
3. **DiscountBondFactory 클래스:**
   - `DiscountBondFactory` 클래스는 할인 채권 객체를 생성하는 팩토리 클래스입니다. `BondFactory`를 상속하고 `createBond` 메서드를 구현합니다.
   - 새로운 할인 채권을 생성하려면 `DiscountBondFactory`를 사용하며, 할인 채권 객체 생성 로직을 캡슐화하여 단순화합니다.
4. **ZeroCouponBondFactory 클래스:**
   - `ZeroCouponBondFactory` 클래스는 이표 채권 객체를 생성하는 팩토리 클래스입니다. `BondFactory`를 상속하고 `createBond` 메서드를 구현합니다.
   - 새로운 이표 채권을 생성하려면 `ZeroCouponBondFactory`를 사용하며, 이표 채권 객체 생성 로직을 캡슐화하여 단순화합니다.

- **Bond 인터페이스:**
  - `Bond` 인터페이스는 모든 채권 클래스가 구현해야 하는 메서드를 정의합니다. 이를 통해 모든 채권 객체에 대한 통일된 인터페이스를 제공합니다.
  - 다양한 종류의 채권 객체를 일관된 방식으로 다룰 수 있으며, 채권 계산에 필요한 메서드를 표준화하여 사용할 수 있습니다.

아래 4개는 Bond를 따르는 채권 클래스입니다.

이 클래스들은 `Bond` 인터페이스를 구현하고 `Bond` 인터페이스를 따르므로 채권의 다양한 기능을 표준화된 방식으로 사용할 수 있습니다.

1. **CompoundBond 클래스:**
   - `CompoundBond` 클래스는 복리 이자 계산 방식을 제공하는 채권 객체를 나타냅니다.
   - 복리 이자 계산 로직을 구체화하여 복리 채권 객체를 생성하고 사용할 수 있습니다.
2. **SimpleBond 클래스:**
   - `SimpleBond` 클래스는 단리 이자 계산 방식을 제공하는 채권 객체를 나타냅니다.
   - 단리 이자 계산 로직을 구체화하여 단리 채권 객체를 생성하고 사용할 수 있습니다.
3. **DiscountBond 클래스:**
   - `DiscountBond` 클래스는 할인 채권을 나타내며 할인 채권에 대한 계산 방식을 제공합니다.
   - 할인 채권 계산 로직을 구체화하여 할인 채권 객체를 생성하고 사용할 수 있습니다.
4. **ZeroCouponBond 클래스:**
   - `ZeroCouponBond` 클래스는 이표 채권을 나타내며 이표 채권에 대한 계산 방식을 제공합니다.
   - 이표 채권 계산 로직을 구체화하여 이표 채권 객체를 생성하고 사용할 수 있습니다.

효과:

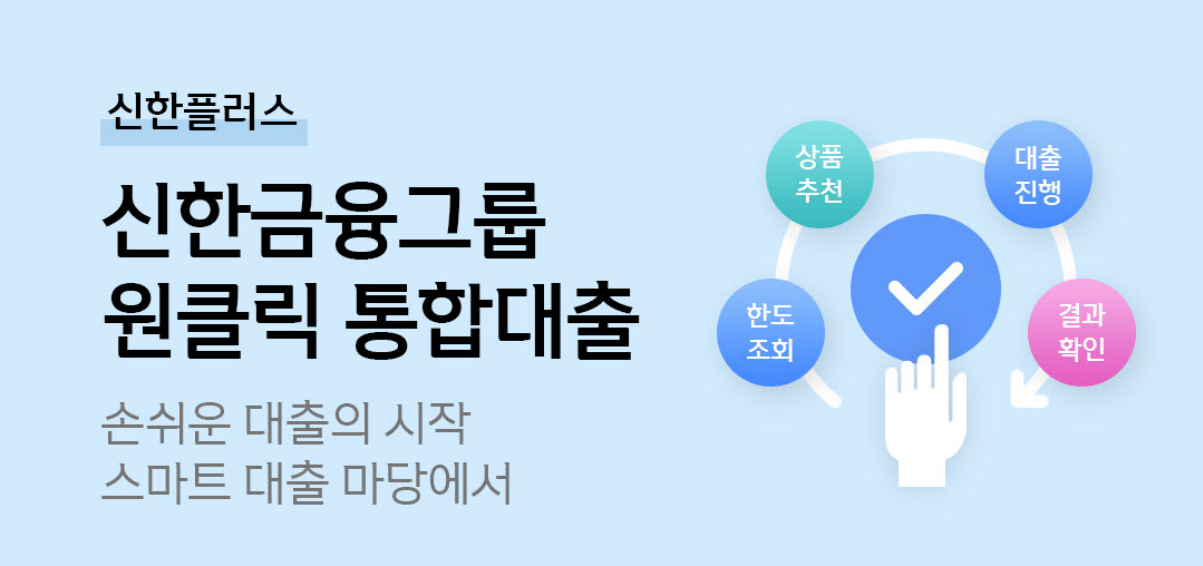
- Factory Method 패턴을 사용하면 새로운 채권 종류를 손쉽게 추가하고, 채권 객체의 생성 및 사용을 표준화할 수 있습니다.
- 각 채권 클래스는 자체적으로 계산 방식을 구현하므로, 계산 로직을 분리하여 관리 및 확장이 가능합니다.
- 인터페이스를 사용하여 모든 채권 객체에 일관된 메서드를 적용하여, 클라이언트 코드의 유지 보수 및 확장성을 향상시킵니다.
- 클라이언트 코드에서는 채권을 생성할 때 적절한 `BondFactory` 서브클래스를 선택하고 `createBond` 메서드를 호출하여 채권을 생성합니다. 이로써 다양한 채권 유형을 생성하고 확장하기 쉽게 됩니다.

Facade를 통한 고객 채권 투자 리포트 생성 간편화

Facade 소개

퍼사드 패턴은 복잡한 라이브러리, 프레임워크 또는 클래스들의 집합에 대한 단순화된 인터페이스를 제공하는 구조적 디자인 패턴입니다. 이 패턴은 하위 시스템에 대한 간단한 인터페이스를 제공하여 코드를 단순화하고 유지 보수를 용이하게 합니다.

Facade 실생활 예시



사실 퍼사드 패턴을 가장 쉽게 설명하려면 원클릭 버튼이 가장 손 쉬운 예시입니다.

신한금융그룹 내에서 퍼사드의 사례를 찾아본다면 위에 이미지처럼 대출에도 다양한 과정이 있지만, 원클릭으로 쉽게 이루어지는 신한금융그룹 원클릭 통합대출이 그 예가 아닌가 싶습니다.

버튼 안에 수 많은 과정이 있지만, 버튼 한번으로 모든게 이루어지는, 그게 바로 Facade 패턴이라고 생각합니다.

Facade 장점

- 복잡한 하위 시스템에 대한 간단한 인터페이스를 제공하여 코드를 단순화하고 유지 보수를 용이하게 합니다.
- 클라이언트들이 중요하게 생각하는 기능들만 포함하여 제한된 기능성을 제공합니다.
- 추가적인 퍼사드 클래스를 생성하여 하나의 퍼사드를 관련 없는 기능들로 오염시켜 복잡한 구조로 만드는 것을 방지할 수 있습니다.

Facade를 통한 고객 채권 투자 리포트 생성

Facade를 통해 채권 투자 고객이 reportAll함수만으로 한번에 채권에 관한 본인 정보를 볼 수 있도록 캡슐화 시켰습니다.

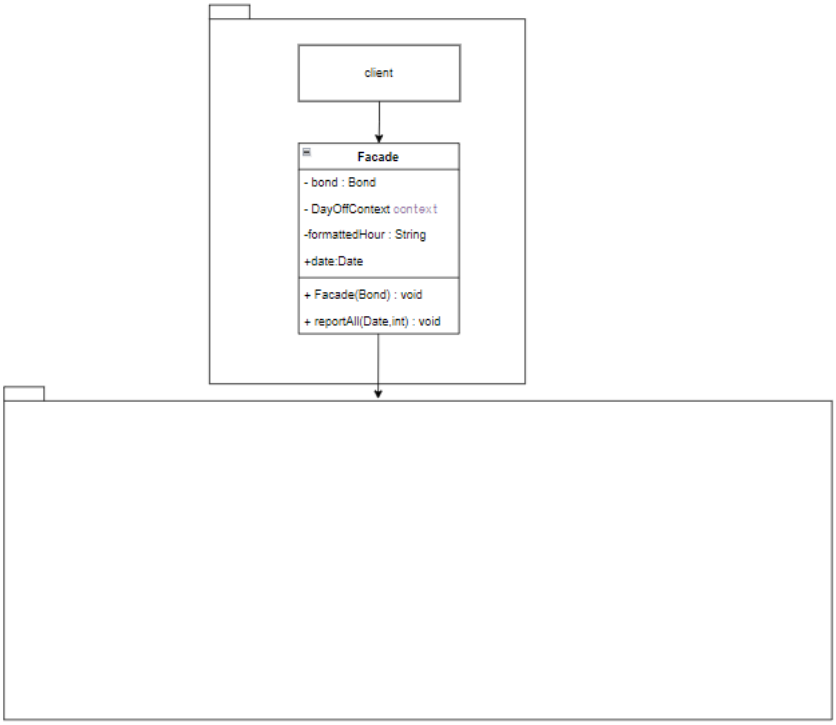
Facade 패턴 도입 전

```
bond.calculateSimpleInterestAmount();
        bond.calculatePreTaxPrincipalAmount();
        bond.calculateYield();
        bond.calculateRemainMaturity();
        bond.calculateInvestmentPeriod();
        bond.calculateYearInterestPayment();
        checkMarketStatus();
```

기존 코드에는 위에 코드에서 보이는 매서드들인 고객 채권 투자 리포트 생성에 필요한 여러 클래스들을 직접 다루어야 했으며, 이는 복잡한 종속성 관계와 함께 코드를 작성하게 만들었습니다. 이와 같은 코드는 코드의 가독성과 유지보수성이 저하되는 현상으로 이어집니다.

Facade 패턴 도입 후

Facade 패턴을 이용하여 아래와 같은 UML 클래스 다이어그램 코드를 작성했습니다.



Facade 클래스를 통하여

서브시스템을 reportAll()을 통해 캡슐화 시켜서 고객에게 간편한 인터페이스를 제공하도록 했습니다.

이를 통해 복잡성이 감소하고 클라이언트 코드가 간소화된 효과를 볼 수 있었습니다

프로젝트를 마치며

이 밖에도 과세 유형이나, 채권 시장의 종류 등 디자인패턴을 적용할 다양한 분야가 있었지만, 채권에 대한 전문적인 지식과 디자인패턴에 대한 실력이 부족하여 더 많은 기능을 만들지 못한 것이 아쉽습니다.

하지만 이 프로젝트를 통해 채권 시장에 디자인 패턴을 적용해야할 필요성을 느끼게 되었습니다.